EPFL, CIVIL-127

Programming and software development for engineers

Lambdas

```
lambda x: len(x[1])


def f1(x):
    return len(x[1])
```

- Lambdas are anonymous functions with an implied return
- No parenthesis around parameters
- This lambda takes one parameter (`x`) and returns the length of `x[1]` (if x is a tuple, it'll return len of the second element)
- This lambda is equivalent to `f1`

Lambdas

```
foo(lambda x: len(x[1]), ...)
```

- Lambdas are mostly used as arguments to specific functions/methods

Lambdas

```
l = sorted([(1, "blue"),
            (4, "red"),
            (10, "green")],
           key=lambda x: len(x[1]))
print(l)  # [(4, 'red'), (1, 'blue'),
(10, 'green')]
```

- Useful e.g. with `sort` and `sorted` to customize the sorting behavior
- Note: `sorted` is like `sort` but returns a new list instead of mutating the existing list

Lambdas

```
normalize_case = lambda s: s.casefold()
```

- Don’t use if you are going to assign the lambda to a variable. Use a regular function definition instead

Lambdas

```
l = sorted(["blue", "red", "green"],
  key=len)
print(l) # ['red', 'blue', 'green']
```

- Don’t use a lambda when you can simply use the function name or operator (see operators module)

Lambdas

- Don’t use if writing the code on multiple lines will be easier to read
- Lambdas generally hurt
  - Readability
  - Testability
  - Reusability
- Weigh the pros/cons before using them

Lambdas

```
s = []
for x in range(5):
    s.append(lambda: x**2)

for x in range(5):
    s.append(lambda n=x: n**2)

for i in s:
    print(i())
```

- Lambdas capture scope, and you can control when the variable will get bound
- Output:

```
16
16
16
16
16
0
1
4
9
16
```

Control Flow

- if
- for
- while
- match

Control flow enables automation – the core reason for writing programs.

if, elif, elif, elif, else

```
if some_condition:
    ... # this will run if some_condition is truthy
... # this will usually always run
```

- You can have just an `if` statement

if, elif, elif, elif, else

```
if some_condition:
    ... # this will run if some_condition is truthy
else:
    ... # this will run if some_condition is fasly
... # this will always run
```

- You can have an `if` statement with an `else` branch

if, elif, elif, elif, else

```
if color == "blue":
 ...
else:
 if color == "red":
   ...
 else:
   if color == "green":
     ...
   else:
     ...
```

- You can put `if` statement inside `else` branches, but your code can become hard to read

if, elif, elif, elif, else

```
if color == "blue":
    ...
elif color == "red":
    ...
elif color == "green":
    ...
else:
    ...
```

- Use `elif` instead, to lower the indentation and increase readability

if, elif, elif, elif, else

```
# explain we go left when some_condition is true
if some_condition:
    ...
# explain we go right when some_other_condition is true
elif some_other_condition:
    ...
# more comment explaining yet_another_condition
elif yet_another_condition:
    ...
# reject
else:
    ...
```

- Don’t comment above the conditional statements
- Put your comments inside the `if`/`elif`/`else` statements, reducing redundancy

if, elif, elif, elif, else

```
if some_condition:
    # go left
    ...
elif some_other_condition:
    # go right
    ...
elif yet_another_condition:
    # do something
    ...
else:
    # reject
    ...
```

- Comment your code inside the `if`/`elif`/`else` branches to remove redundancy

if, elif, elif, elif, else

```
def foo():
    if some_condition:
        ...
        return

    # else is implied
    if some_other_condition:
        ...
        return
    ...
```

- When you `return` from the first `if` block, you don't need an `elif` or `else` block.

Conditional Expression

```
a = 1

b = "foo" if a == 1 else "bar"

print(b) # foo
```

- It's like an `if`/`else` but as an expression
- Some programmers like this feature

Nested Loops

```
for i in range(10):
    for j in range(10):
        if j == i:
            continue
        if j + i == 10:
            break
        print(i, j)
    if i % 3 == 0:
        continue
```

- `break` and `continue`, to break or continue iterating
- You can't break or continue the outer loop from within the inner loop, you must refactor your code

Nested Loops

```
def foobar():
    for i in range(10):
        for j in range(10):
            if j == i:
                continue
            if j + i == 10:
                return
            print(i, j)
        if i % 3 == 0:
            continue
```

- You can't break or continue the outer loop from within the inner loop, you must refactor your code and use a `return` statement

while loops

```
def prompt_yes_or_no(prompt=""):
    while True:
        a = input(prompt)
        if a == "yes" or a == "no":
            return a
```

- If you know how many times you want to loop, use `for`
- Otherwise, use `while`
- You can use `break` and `continue` with `while` loops in a way similar to what you can do with `for` loops

match

```
match input():
    case "w":
        ...
    case "d":
        ...
    case "a":
        ...
    case "s":
        ...
```

- Replaces many `if`, `elif`, `elif`, `else` statements

match

```
match input():
    case "w" | "i":
        ...
    case "d" | "k":
        ...
    case "a" | "j":
        ...
    case "s" | "l":
        ...
```

- Combine multiple cases with `|` (acts like an `or` expression)

match

```
a = ((1, 2), "foo")

match a:
    case ((1, _), "bar"):
        print("1.")
    case ((x, 2), "foo"):
        print("2.", x)
    case _:
        print(a)
```

- You can destructure tuples
- First matching case wins
- `_` in the case statement means match anything
- `x` in the case statement means match anything and assign to `x`.
- Use `case _:` to handle the default case
- More info https://peps.python.org/pep-0636/

Error handling: error value

- Functions and methods can indicate errors by returning an error value. E.g. empty, None, or one or more error states (see sokoban solutions exercise_3_3)
  - Callers must then check the return value and handle the error case. It is common for callers to then propagate the error value.

Error handling: error value

```
from math import sqrt

def quadratic(a, b, c):
    """
    Calculates both solutions to ax^2 + bx +
c == 0, where a, b, and c are numbers.
    Returns None or a 2-tuple with each
solution.
    """

    # Check if a solution exists
    d = b ** 2 - 4 * a * c
    if d < 0:
        return None
    # Compute d's square root only once
    sd = sqrt(d)
    s1 = (-b - sd) / (2 * a)
    s2 = (-b + sd) / (2 * a)
    return s1, s2

print(quadratic(1, -7, 12))  # (3.0, 4.0)
print(quadratic(1, -7, 50))  # None
```

Error handling: error value

```
def closest(x, a, b, c):
    """
    Returns closest point p to x such that
ap^2+ap+c == 0 or None.
    """
    r = quadratic(a, b, c)
    if r == None:
        return None

    s1, s2 = r
    if abs(x - s1) < abs(x - s2):
        return s1
    return s2
```

- Error handling with error values
  - Error handling code is needed everywhere
  - You might forget to handle an error case

Error handling: exceptions

```
from math import sqrt

def quadratic(a, b, c):
    """
    Calculates both solutions to ax^2 + bx +
c == 0, where a, b, and c are numbers.
    Returns a 2-tuple with each solution or
raises ValueError.
    """

    # Check if a solution exists
    d = b ** 2 - 4 * a * c
    if d < 0:
        raise ValueError("no solutions")
    # Compute d's square root only once
    sd = sqrt(d)
    s1 = (-b - sd) / (2 * a)
    s2 = (-b + sd) / (2 * a)
    return s1, s2

print(quadratic(1, -7, 12))   # (3.0, 4.0)
print(quadratic(1, -7, 50))   # ValueError
```

Error handling: exceptions

```
def closest(x, a, b, c):
    """
    Returns closest point p to x
such that ap^2+ap+c == 0 or raises
ValueError.
    """
    s1, s2 = quadratic(a, b, c)
    if abs(x - s1) < abs(x - s2):
        return s1
    return s2
```

- Error handling with exceptions
  - Callers can let the exception propagate
  - You should document the fact that an exception can happen
  - This documentation is almost good: we forgot to mention the code will raise ZeroDivisionError if parameter a is zero
  - We also don't document what happens if both solutions are equidistant

Error handling: exceptions

```
def foo():
    try:
        closest(10, 1, -7, 50)
    except ValueError:
        print("no closest point")
```

- Error handling with exceptions
  - Any caller in the call chain can handle the exception
  - except tells the runtime which specific exception you want to handle
  - Use `try:` and `except <name>:`

Error handling: exceptions

```
class MyException(Exception):
    pass

def foo():
    try:
        raise MyException()
        ...
    except Exception as e:
        traceback.print_exception(e)
```

- `ValueError` isn't special, you can use any of the predefined exceptions or create your own
- See standard exceptions
- Use `except Exception:` to catch most exceptions

Error handling: exceptions

```
class MyException(Exception):
   pass

def foo():
   try:
       raise IndexError("12 is out of bounds")
       ...
   except Exception as e:
       traceback.print_exception(e)
```

- `ValueError` isn't special, you can use any of the predefined exceptions or create your own
- See standard exceptions
- Use `except Exception:` to catch most exceptions

Error handling: exceptions

```
class MyException(Exception):
   pass

def foo():
   try:
       raise KeyError()
       ...
   except Exception as e:
       traceback.print_exception(e)
```

- `ValueError` isn't special, you can use any of the predefined exceptions or create your own
- See standard exceptions
- Use `except Exception:` to catch most exceptions

Error handling: exceptions

```
raise Foo

raise Foo()

raise Foo("some message")
```

- You can raise a class, which is equivalent to raising an instance

Handling multiple exceptions

```
def foo():
    try:
        ...
    except MyException:
        ... handle MyException ...
    except IOError:
        ... handle IOError(e.g. file not found) ...
```

- Use multiple `except` blocks to handle different exceptions
- There's a hierarchy of exceptions, which should help with error handling

Catching all

```
def foo():
    try:
        ...
    except BaseException:
        ...
```

- If you `except BaseException`, you'll catch all the exceptions. Including someone trying to quit your application with `ctrl-c`
- You almost never want to do this

Catching all

```
def foo():
    try:
        ...
    except:
        ...
```

- `except` without an exception type also catches all exceptions
- You almost never want to do this

Catching all

```
def foo():
    try:
        ...
    except Exception:
        ...
```

- If you `except Exception`, you'll catch most exceptions
- Usually, you'll want to be more precise but catching most exceptions is useful in some cases

Finally

```
def foo():
    try:
        print(1)
        raise Exception()
    finally:
        print(2)

foo()
```

- Finally always runs
- Enables cleaning things up
- This code prints 1, 2, and then an exception with a stack trace

Finally

```
def foo():
    try:
        print(1)
        raise Exception()
    except Exception:
        print(2)
    finally:
        print(3)

foo()
```

- Finally always runs
- Enables cleaning things up
- This code prints 1, 2, 3

Finally

```
def foo(n):
    try:
        print(1)
        if n == 2:
            raise Exception()
        return n * 2
    except Exception:
        print(2)
        return n * 3
    finally:
        print(3)
        return n * 4

print("one:", foo(1))
print("two:", foo(2))
```

- If there are multiple `return` statements, the value from finally is returned
  - Don't write convoluted code like this!
- Output:
  1
  3
  one: 4
  1
  2
  3
  two: 8

Simulating a `for` loop with a `while` loop

```
a = range(10)


iterator = a.__iter__()
try:
    while True:
        n = iterator.__next__()
        print(n)
except StopIteration:
    pass
```

- Please don’t write code like this

Simulating a `while` loop with a `for` loop

```
class Cond:
   def __init__(self, expr):
       self.expr = expr

   def __iter__(self):
       return self

   def __next__(self):
       if not self.expr():
           raise StopIteration

s = 0
n = 0
for _ in Cond(lambda: s < 10):
   print(n)
   n += 1
   s += n
```

- Nor code like this

with

```
with open(xsb_file, "r") as f:
  ...
```

- The `with` construct is related to exceptions
- See PEP-343

with

```
class Foobar():
    def __init__(self, id):
        self.id = id
        print(self.id, "__init__ called")

    def __enter__(self):
        print(self.id, "__enter__ called")
        return self

    def __exit__(self):
        print(self.id, "__exit__ called")

with Foobar(1) as f:
    print("inside with")

with Foobar(2) as f:
    print("inside with")
    raise Exception()
```

- `__enter__` and `__exit__` are always called, even if there's an exception inside the with block
- This makes `with` construct useful for anything which requires cleaning up (e.g. closing a file after we are done using it)

Assertions

- Use assertions to express assumptions
- When an assert is incorrect, an exception will be raised
- Assertions are a cheap way to make code more readable and reduce likelihood of bugs

Assertions

```python
def foobar(x):
    assert x != 0
    print(10 / x)
```

```python
def foobar(x):
    if x == 0:
        raise AssertionError
    print(10 / x)
```

These two pieces of code are roughly equivalent. Left side is more readable

Assertions

- Assertions are useful when handling external data. E.g. in Sokoban, while loading the level:
  - Assert that there's only one player in the level file
  - Assert that there is an equal number of boxes and goals (equal or more also works)
  - Assert that there are no unknown characters in the file
  - Assert that each line is the same width (or handle the case if it's not)
  - Etc
- Assertions are useful for ensuring your internal state is consistent. E.g. (depending on your state representation) you can assert that the player is not on top of a box.
- Implement your assertions while you are writing your code (that's when you are thinking about your assumptions) – don't try to add them later

Assertions

```python
from enum import Enum

class Dir(Enum):
    UP = "up"
    DOWN = "down"
    LEFT = "left"
    RIGHT = "right"

match dir:
    case Dir.UP:
        ...
    case Dir.DOWN:
        ...
    case Dir.LEFT:
        ...
    case Dir.RIGHT:
        ...
    case _:
        assert False  # unreachable
```

- `assert False # unreachable` is useful to indicate that a line of code should never be reached
- Useful in `match` statements

Assertions

```
if len(a) == 0 and len(b) == 0:
    ...
    return
if len(a) == 0:
    assert len(b) != 0
    ...
    return
if len(b) == 0:
    assert len(a) != 0
    ...
    return

assert len(a) != 0 and len(b) != 0
...
```

- Very useful in complicated `if` statements
- The code here is (simplified) from a piece of real world code